**Linux五种IO模型性能分析**

[socket阻塞与非阻塞，同步与异步](http://blog.csdn.net/hguisu/article/details/7453390)  
1. 概念理解

     在进行网络编程时，我们常常见到同步(Sync)/异步(Async)，阻塞(Block)/非阻塞(Unblock)四种调用方式：  
同步：  
      所谓同步，就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。**也就是必须一件一件事做**,等前一件做完了才能做下一件事。

例如普通B/S模式（同步）：提交请求->等待服务器处理->处理完毕返回 这个期间客户端浏览器不能干任何事

异步：  
      异步的概念和同步相对。当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。

     例如 ajax请求（异步）: 请求通过事件触发->服务器处理（这是浏览器仍然可以作其他事情）->处理完毕

**阻塞**  
     阻塞调用是指调用结果返回之前，当前线程会被挂起（线程进入非可执行状态，在这个状态下，cpu不会给线程分配时间片，即线程暂停运行）。函数只有在得到结果之后才会返回。

     有人也许会把阻塞调用和同步调用等同起来，实际上他是不同的。对于同步调用来说，很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。 例如，我们在socket中调用recv函数，如果缓冲区中没有数据，这个函数就会一直等待，直到有数据才返回。而此时，当前线程还会继续处理各种各样的消息。

**非阻塞**  
      非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。  
对象的阻塞模式和阻塞函数调用  
对象是否处于阻塞模式和函数是不是阻塞调用有很强的相关性，但是并不是一一对应的。阻塞对象上可以有非阻塞的调用方式，我们可以通过一定的API去轮询状 态，在适当的时候调用阻塞函数，就可以避免阻塞。而对于非阻塞对象，调用特殊的函数也可以进入阻塞调用。函数select就是这样的一个例子。

1. 同步，就是我调用一个功能，该功能没有结束前，我死等结果。  
2. 异步，就是我调用一个功能，不需要知道该功能结果，该功能有结果后通知我（回调通知）  
3. 阻塞，      就是调用我（函数），我（函数）没有接收完数据或者没有得到结果之前，我不会返回。  
4. 非阻塞，  就是调用我（函数），我（函数）立即返回，通过select通知调用者

同步IO和异步IO的区别就在于：数据拷贝的时候进程是否阻塞！

阻塞IO和非阻塞IO的区别就在于：应用程序的调用是否立即返回！

对于举个简单c/s 模式：

同步：提交请求->等待服务器处理->处理完毕返回这个期间客户端浏览器不能干任何事  
异步：请求通过事件触发->服务器处理（这是浏览器仍然可以作其他事情）->处理完毕

同步和异步都只针对于本机SOCKET而言的。

同步和异步,阻塞和非阻塞,有些混用,其实它们完全不是一回事,而且它们修饰的对象也不相同。  
阻塞和非阻塞是指当进程访问的数据如果尚未就绪,进程是否需要等待,简单说这相当于函数内部的实现区别,也就是未就绪时是直接返回还是等待就绪;

而同步和异步是指访问数据的机制,同步一般指主动请求并等待I/O操作完毕的方式,当数据就绪后在读写的时候必须阻塞(区别就绪与读写二个阶段,同步的读写必须阻塞),异步则指主动请求数据后便可以继续处理其它任务,随后等待I/O,操作完毕的通知,这可以使进程在数据读写时也不阻塞。(等待"通知")

1. Linux下的五种I/O模型

1)阻塞I/O（blocking I/O）  
2)非阻塞I/O （nonblocking I/O）  
3) I/O复用(select 和poll) （I/O multiplexing）  
4)信号驱动I/O （signal driven I/O (SIGIO)）  
5)异步I/O （asynchronous I/O (the POSIX aio\_functions)）

前四种都是同步，只有最后一种才是异步IO。

阻塞I/O模型：

        简介：进程会一直阻塞，直到数据拷贝完成

     应用程序调用一个IO函数，导致应用程序阻塞，等待数据准备好。 如果数据没有准备好，一直等待….数据准备好了，从内核拷贝到用户空间,IO函数返回成功指示。

阻塞I/O模型图：在调用recv()/recvfrom（）函数时，发生在内核中等待数据和复制数据的过程。



    当调用recv()函数时，系统首先查是否有准备好的数据。如果数据没有准备好，那么系统就处于等待状态。当数据准备好后，将数据从系统缓冲区复制到用户空间，然后该函数返回。在套接应用程序中，当调用recv()函数时，未必用户空间就已经存在数据，那么此时recv()函数就会处于等待状态。

     当使用socket()函数和WSASocket()函数创建套接字时，默认的套接字都是阻塞的。这意味着当调用Windows Sockets API不能立即完成时，线程处于等待状态，直到操作完成。

    并不是所有Windows Sockets API以阻塞套接字为参数调用都会发生阻塞。例如，以阻塞模式的套接字为参数调用bind()、listen()函数时，函数会立即返回。将可能阻塞套接字的Windows Sockets API调用分为以下四种:

    1．输入操作： recv()、recvfrom()、WSARecv()和WSARecvfrom()函数。以阻塞套接字为参数调用该函数接收数据。如果此时套接字缓冲区内没有数据可读，则调用线程在数据到来前一直睡眠。

    2．输出操作： send()、sendto()、WSASend()和WSASendto()函数。以阻塞套接字为参数调用该函数发送数据。如果套接字缓冲区没有可用空间，线程会一直睡眠，直到有空间。

    3．接受连接：accept()和WSAAcept()函数。以阻塞套接字为参数调用该函数，等待接受对方的连接请求。如果此时没有连接请求，线程就会进入睡眠状态。

   4．外出连接：connect()和WSAConnect()函数。对于TCP连接，客户端以阻塞套接字为参数，调用该函数向服务器发起连接。该函数在收到服务器的应答前，不会返回。这意味着TCP连接总会等待至少到服务器的一次往返时间。

　　使用阻塞模式的套接字，开发网络程序比较简单，容易实现。当希望能够立即发送和接收数据，且处理的套接字数量比较少的情况下，使用阻塞模式来开发网络程序比较合适。

    阻塞模式套接字的不足表现为，在大量建立好的套接字线程之间进行通信时比较困难。当使用“生产者-消费者”模型开发网络程序时，为每个套接字都分别分配一个读线程、一个处理数据线程和一个用于同步的事件，那么这样无疑加大系统的开销。其最大的缺点是当希望同时处理大量套接字时，将无从下手，其扩展性很差

非阻塞IO模型

       简介：非阻塞IO通过进程反复调用IO函数（多次系统调用，并马上返回）；在数据拷贝的过程中，进程是阻塞的；

       我们把一个SOCKET接口设置为非阻塞就是告诉内核，当所请求的I/O操作无法完成时，不要将进程睡眠，而是返回一个错误。这样我们的I/O操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，会大量的占用CPU的时间。

    把SOCKET设置为非阻塞模式，即通知系统内核：在调用Windows Sockets API时，不要让线程睡眠，而应该让函数立即返回。在返回时，该函数返回一个错误代码。图所示，一个非阻塞模式套接字多次调用recv()函数的过程。前三次调用recv()函数时，内核数据还没有准备好。因此，该函数立即返回WSAEWOULDBLOCK错误代码。第四次调用recv()函数时，数据已经准备好，被复制到应用程序的缓冲区中，recv()函数返回成功指示，应用程序开始处理数据。



     当使用socket()函数和WSASocket()函数创建套接字时，默认都是阻塞的。在创建套接字之后，通过调用ioctlsocket()函数，将该套接字设置为非阻塞模式。Linux下的函数是:fcntl().  
    套接字设置为非阻塞模式后，在调用Windows Sockets API函数时，调用函数会立即返回。大多数情况下，这些函数调用都会调用“失败”，并返回WSAEWOULDBLOCK错误代码。说明请求的操作在调用期间内没有时间完成。通常，应用程序需要重复调用该函数，直到获得成功返回代码。

    需要说明的是并非所有的Windows Sockets API在非阻塞模式下调用，都会返回WSAEWOULDBLOCK错误。例如，以非阻塞模式的套接字为参数调用bind()函数时，就不会返回该错误代码。当然，在调用WSAStartup()函数时更不会返回该错误代码，因为该函数是应用程序第一调用的函数，当然不会返回这样的错误代码。

    要将套接字设置为非阻塞模式，除了使用ioctlsocket()函数之外，还可以使用WSAAsyncselect()和WSAEventselect()函数。当调用该函数时，套接字会自动地设置为非阻塞方式。

　　由于使用非阻塞套接字在调用函数时，会经常返回WSAEWOULDBLOCK错误。所以在任何时候，都应仔细检查返回代码并作好对“失败”的准备。应用程序连续不断地调用这个函数，直到它返回成功指示为止。上面的程序清单中，在While循环体内不断地调用recv()函数，以读入1024个字节的数据。这种做法很浪费系统资源。

    要完成这样的操作，有人使用MSG\_PEEK标志调用recv()函数查看缓冲区中是否有数据可读。同样，这种方法也不好。因为该做法对系统造成的开销是很大的，并且应用程序至少要调用recv()函数两次，才能实际地读入数据。较好的做法是，使用套接字的“I/O模型”来判断非阻塞套接字是否可读可写。

    非阻塞模式套接字与阻塞模式套接字相比，不容易使用。使用非阻塞模式套接字，需要编写更多的代码，以便在每个Windows Sockets API函数调用中，对收到的WSAEWOULDBLOCK错误进行处理。因此，非阻塞套接字便显得有些难于使用。

    但是，非阻塞套接字在控制建立的多个连接，在数据的收发量不均，时间不定时，明显具有优势。这种套接字在使用上存在一定难度，但只要排除了这些困难，它在功能上还是非常强大的。通常情况下，可考虑使用套接字的“I/O模型”，它有助于应用程序通过异步方式，同时对一个或多个套接字的通信加以管理。

IO复用模型：

             简介：主要是select和epoll；对一个IO端口，两次调用，两次返回，比阻塞IO并没有什么优越性；关键是能实现同时对多个IO端口进行监听；

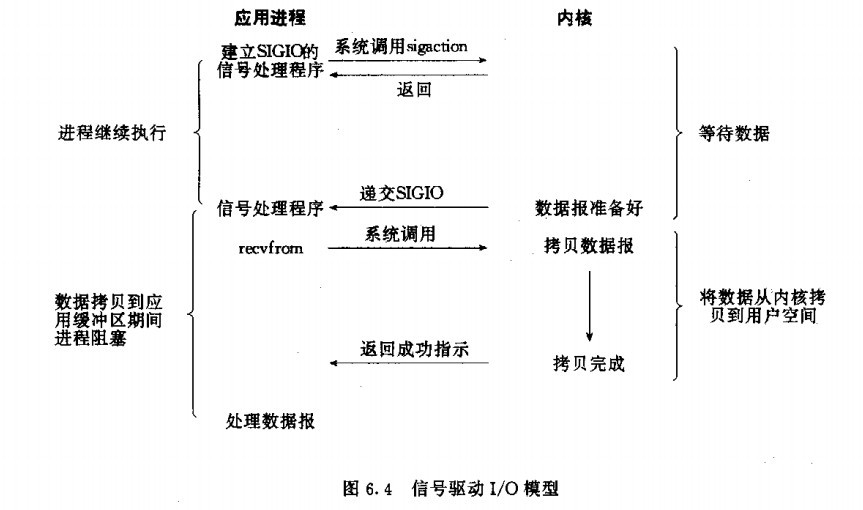
      I/O复用模型会用到select、poll、epoll函数，这几个函数也会使进程阻塞，但是和阻塞I/O所不同的的，这两个函数可以同时阻塞多个I/O操作。而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时，才真正调用I/O操作函数。



信号驱动IO

    简介：两次调用，两次返回；

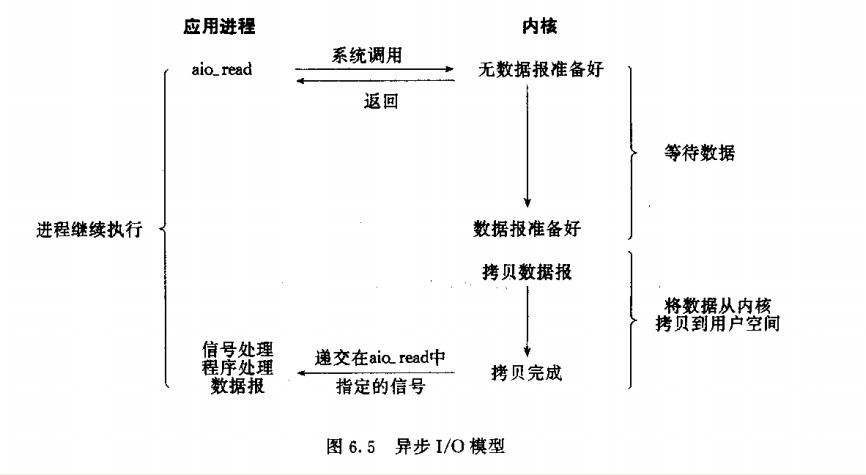
    首先我们允许套接口进行信号驱动I/O,并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个SIGIO信号，可以在信号处理函数中调用I/O操作函数处理数据。



异步IO模型

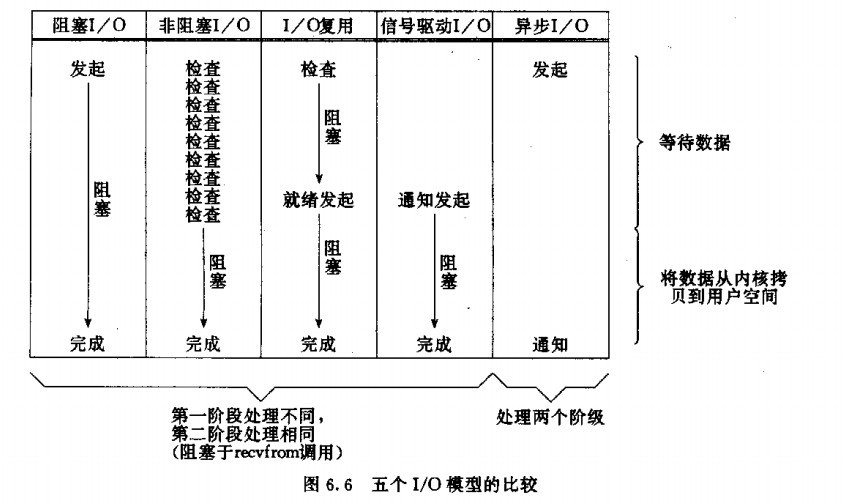
         简介：数据拷贝的时候进程无需阻塞。

当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者的输入输出操作



同步IO引起进程阻塞，直至IO操作完成。  
异步IO不会引起进程阻塞。  
IO复用是先通过select调用阻塞。

**5个I/O模型的比较：**



1. select、poll、epoll简介

epoll跟select都能提供多路I/O复用的解决方案。在现在的Linux内核里有都能够支持，其中epoll是Linux所特有，而select则应该是POSIX所规定，一般操作系统均有实现

select：

select本质上是通过设置或者检查存放fd标志位的数据结构来进行下一步处理。这样所带来的缺点是：

1、 单个进程可监视的fd数量被限制，即能监听端口的大小有限。

      一般来说这个数目和系统内存关系很大，具体数目可以cat /proc/sys/fs/file-max察看。32位机默认是1024个。64位机默认是2048.

2、 对socket进行扫描时是线性扫描，即采用轮询的方法，效率较低：

       当套接字比较多的时候，每次select()都要通过遍历FD\_SETSIZE个Socket来完成调度,不管哪个Socket是活跃的,都遍历一遍。这会浪费很多CPU时间。如果能给套接字注册某个回调函数，当他们活跃时，自动完成相关操作，那就避免了轮询，这正是epoll与kqueue做的。

3、需要维护一个用来存放大量fd的数据结构，这样会使得用户空间和内核空间在传递该结构时复制开销大

poll：

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。这个过程经历了多次无谓的遍历。

它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点：

1、大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。                                                                                                                                      2、poll还有一个特点是“水平触发”，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。

epoll:

epoll支持水平触发和边缘触发，最大的特点在于边缘触发，它只告诉进程哪些fd刚刚变为就需态，并且只会通知一次。还有一个特点是，epoll使用“事件”的就绪通知方式，通过epoll\_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll\_wait便可以收到通知

epoll的优点：

1、没有最大并发连接的限制，能打开的FD的上限远大于1024（1G的内存上能监听约10万个端口）；  
**2、效率提升**，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数；  
      即Epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。

3、 内存拷贝，利用mmap()文件映射内存加速与内核空间的消息传递；即epoll使用mmap减少复制开销。  
  
select、poll、epoll 区别总结：

1、支持一个进程所能打开的最大连接数

|  |  |
| --- | --- |
| select | 单个进程所能打开的最大连接数有FD\_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是32\*32，同理64位机器上FD\_SETSIZE为32\*64），当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试。 |
| poll | poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的 |
| epoll | 虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接 |

2、FD剧增后带来的IO效率问题

|  |  |
| --- | --- |
| select | 因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”。 |
| poll | 同上 |
| epoll | 因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。 |

3、 消息传递方式

|  |  |
| --- | --- |
| select | 内核需要将消息传递到用户空间，都需要内核拷贝动作 |
| poll | 同上 |
| epoll | epoll通过内核和用户空间共享一块内存来实现的。 |

总结：

综上，在选择select，poll，epoll时要根据具体的使用场合以及这三种方式的自身特点。

1、表面上看epoll的性能最好，但是在连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多函数回调。

2、select低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善